A Study of Program Behavior to Establish Temporal Locality at the Function Level

Ravi Batchu Saul Levy Miles Murdocca

Department of Computer Science
Rutgers University
{batchu,levy,murdocca}@cs.rutgers.edu

Abstract

The trend in computer architecture is that processor speeds are increasing rapidly compared to memory access times and the relatively stagnant disk speed. Computer software, on the other hand is characterized by growing program sizes and sophisticated functionality. The combination of these factors has resulted in a processor memory bottleneck, which is worsening with time. While program behavior has been studied at page level and cache level and the locality at page, cache and *block* levels has been exploited, there has been comparatively much lesser amount of work to exploit locality at the level of functions, and no prior work to study program behavior at this level.

In this paper we show that there is a considerable amount of temporal locality at the level of functions. In particular we show that a working set of functions containing 40% of all the program's functions results in a miss rate of less than 2%. Moreover, we observe that, in almost all cases, even working sets having half as many functions result in similar low miss rates. Our experiments indicate that program execution is characterized by a working set of functions which changes with time and a *thrashing* like phenomenon results when the *function footprint* is not resident in memory.

Keywords: Program Behavior, Temporal Locality, Working Set, Thrashing, Function/Procedure Re-ordering

1 Introduction

The predominant style of programming in the industry has been the modular approach in which code is written as several functions. The problem, which needs to be solved, is broken down into smaller problems, which are broken down into smaller sub-problems and so on. These smaller pieces of code which result from this top down decomposition process are referred to as procedures, functions or subroutines depending on the language used for programming. The intent of this paper is to report on a study of the usage of functions during the execution of a program. We believe that a small set of functions is referenced at any particular point of execution and that this set changes slowly with execution. Such aspects of references to functions are similar to those found in references to pages and perhaps even more pronounced. We describe the experiments and analysis we performed to establish this characteristic of program behavior. In theory, as in the case of memory pages, it is possible to write programs, which do not exhibit this behavior. However, that is rarely the case in practice.

The page is considerably bigger than a typical function size. Page size has been determined by the following factors:

i. Transfer from the disk into physical memory is slow compared to the physical memory access and there is a fixed overhead of the seek

time for every disk access. A single transfer of a large page is more desirable than multiple transfers of smaller pages from the disk.

ii. Smaller page size results in larger page tables. The page size has to be reasonably large so that the overhead of the additional space for the page tables is only a small fraction of the amount of space covered by the pages.

iii. Larger page size results in fewer TLB misses.

iv. Larger page size results in internal fragmentation. Some of the portions of the page brought in may not be needed.

With time we have observed that the improvements in the semiconductor technology have resulted in faster CPUs and memory chips. However, due to the inherent mechanical nature of the disks, the disk access time has not shown such improvements. This means that the affect of the factor (i) is increasing and is likely to increase further. The introduction of 64 bit processors with a larger address space would require larger page tables with the current page size. This would increase the affect of (ii) and also influence the page size to increase. So it is unlikely that we are going to witness a decrease in the page size.

The above factors have resulted in a page size which is typically around 4K. Function size is normally much smaller than say 500 bytes. Intuitively, this can be explained by the fact that the problems which programmers are trying to solve are broken down into fairly small pieces for ease of implementation. At the high level, functions have limited amount of code, say a screenful of lines. Human nature is unlikely to be subject to as much change as the architectural characteristics. Therefore, this disparity in sizes of functions and pages is bound to remain with time.

Besides the fact that the existence of a working set of functions during program execution is an important characteristic of program behavior, there are very important implications of this realization. There is a growing bottleneck between process speeds and memory access times, which is alleviated by the organization of memory into a hierarchy of different kinds to keep the price/performance ratio at a viable level. The existence of a working set of functions can be exploited by caching frequently used functions in fewer pages. This could be performed by the system software on current day machines, or alternatively by a more radical change in the architecture to dynamically accommodate the frequently used functions in faster memory.

We performed a study of some applications by building a simulation platform to non-intrusively find out the sequence in which various functions are used. Our experiments establish the properties of locality at the function level by maintaining a working set of functions. Note that the term "working set" has also been used to refer to the footprint of the program, consisting of the set of functions accessed by the program during program execution, as opposed to the set of function being maintained in a separate area for the purpose of our experiments. The precise meaning is obvious from the context of its occurrence.

2 Related Work

To our knowledge this is the first study of its kind, in which the references to functions are studied. The immediate consequence of temporal locality at the level of functions is to explore the possibility of dynamically optimizing the execution of the *hot* functions - perhaps by packing them together to reduce the working set size and cache conflicts. There has been prior work in changing code layout at the function level at compile time as well as dynamically. There have also been efforts to exploit program locality dynamically at other levels of granularity.

Pettis and Hanson [21], Scott McFarling [19], Hatfield and Gerald [13], Gloy and Smith [15],have presented methods to rearrange the procedures, which comprise the executable, based on profile data to improve memory locality. Most of these use profile data in the form of a *weighted call graph* (WCG). In a WCG, there is a node for each function and an undirected edge connects two nodes U and

V if U calls V or vice-versa. An edge between two nodes has a weight equal to the number of times the functions, represented by the two nodes, call each other. If two functions are placed adjacent in memory, they are less likely to incur conflict misses because would get mapped to different portions of the cache, unless the sum of their sizes is greater than the cache size. The technique used by Pettis and Hanson, iterates over the WCG in the order of decreasing edge weight. For each edge, it tries to place the functions of the edge as close as possible. Unlike the earlier approaches, [15] records temporal ordering information during the profiling stage to get better layouts.

The effectiveness of static code layout techniques depends on how well the initial execution, used for gathering the profile data, reflects the nature of program execution resulting from the different data sets used in practice. Modern applications use Dynamically loaded libraries (DLLs), on the Windows platforms, and shared objects, on the unix platforms. Application software often gets shipped as a collection of DLLs. Dynamic linking imposes limits on compile time code layout strategies. Dynamic code generation environments, like Java JITs, also make static code layout techniques impractical. Unlike the static techniques, the cost of restructuring the code does not get amortized over several executions in the dynamic techniques. Dynamic techniques are, therefore, faced with the challenge of recouping the overhead during each program execution.

Chen and Leupen [5] presented a novel heuristic, called *just in time code layout*, which copies functions into memory, when they are first invoked. Thus the order in which the functions appear in the executing image gets fixed when they get loaded into memory. They reported a reduction in the footprint of the executable by 50%. They also showed that their technique provides improvements in instruction misses which are comparable to those achieved by Pettis and Hansen [21].

Bershad et al. [4] reported a dynamic technique to avoid conflict misses in a direct-mapped cache which is much larger than the page size. Their technique works at the granularity of pages and uses a special hardware device, called the *Cache Miss Lookaside Buffer (CML) buffer*, to detect conflicts. Cache misses for each page get recorded in the CML buffer. When the misses exceed a certain threshold the CML buffer interrupts the CPU. The operating system, then, looks for two or more pages which share the same "cache-page"(portion of the direct-mapped cache to which a page in physical memory is mapped to) and have cache misses higher than a certain threshold. All but one of those pages are copied so that they no longer share the same cache-page.

More recently there have been several efforts involving code transformation during run time, although not at the level of functions, for different purposes. There are systems, referred to as the *dynamic binary translation systems*, which run binaries compiled for one platform on a totally different platform. Crusoe [17] achieves power efficiency by translating the x86 code presented to it and running it on a power efficient VLIW processor which is built with fewer transistors. DAISY [14] achieves high performance by translating *PowerPC* binaries for the DAISY VLIW processor. Shade [8] and Embra [26] provide faster simulation capabilities. Strata [24] is an infrastructure for software binary translation, which can be extended to build translation systems for implementing specific functionality. The key factor in all of these technologies is the judicious choice of sufficiently frequently executed pieces of code for translation into native code, while interpreting the less frequently used portions of the program.

There are also dynamic optimization systems which exploit locality in code to aggressively optimize code at run time. Dynamo [3] and Mojo [6] are both user level software efforts to optimize programs executing on the HPUX/PA-RISC and Windows/x86 platforms, respectively. Replay [20] is a new processor framework that supports dynamic optimization, in hardware. Kistler and Franz [16] presented a dynamic optimization system by utilizing the idle time and dynamically collecting execution profiles.

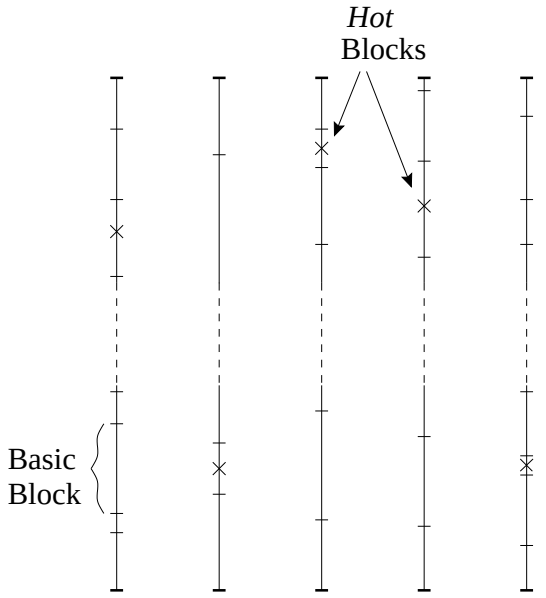

Figure 1: Above figure depicts a program with 5 functions. The *hot* blocks are marked with an 'X'. While a working set of basic blocks would contain only a portion of the code, a working set of functions would contain the entire code. The block level locality does not translate into a function level locality.

The above dynamic optimization and translation systems work by selecting pieces of code at the *basic block* (a maximal sequence of machine instructions which can be entered only at the beginning and exited only at the end) level, and often chaining blocks which get executed one after another at run time to identify *hot* paths. The selected portion of the code is then translated and/or optimized with the expectation that the overhead involved would be more than made up for, by the repeated execution of the optimized code. The success of such research efforts indicates temporal locality at the block level. However, temporal locality at the block level does not necessarily imply temporal locality at the function level.

Consider an example, as shown in Fig 1, in which the hot blocks are spread across all the functions of the program. If we assume that each function, on an average, has 10 blocks and the hot blocks are the only ones which get accessed then only one-tenth of all the blocks would be in a working set consisting of blocks. On the other hand, a working set consisting of functions would require all the functions, consuming much more space. This example demonstrates that locality at the block level does not automatically imply locality at the function level.

The above techniques for dynamically optimizing code involve disassembling the binary executable followed by aggressive optimization of code. Because of the high overhead, they have to be fairly selective to ensure that the code which is optimized would in fact be repeatedly executed in the future. There have been cases for which Dynamo and Morph had to "bail out", i.e., stop optimizing the binary and just execute it normally. We believe that simpler heuristics for function re-ordering, having a lesser overhead, would be less susceptible to such failures. Moreover, function level optimization does not necessarily rule out the optimization techniques used in the above dynamic optimization systems completely. It is possible to use both approaches simultaneously.

The success of schemes which dynamically reorder functions hinges on the existence of a working sets at the function level and methods for identifying them accurately and efficiently. In this paper we report on the experiments which show that programs exhibit a working set nature at the function level. While there is a large body of work on program behavior [25, 11, 12, 10, 22], to the best of our knowledge, there has been no effort to study locality at the level of functions.

3 The Simulation Platform

Program simulation techniques can be classified into one of 3 categories: - trace driven, program driven or execution driven. In the trace driven approach a memory reference trace of the application under study is first collected. The trace is then presented to the simulator in the order in which it was recorded. If the trace is very large, then the disk i/o bandwidth and the amount of free storage space on the disk can be the limiting factors in applying this technique. In addition, if the system be-

ing simulated determines the program control flow, then the simulation becomes unrealistic because the program execution has to be completed before the system is simulated. In the program driven approach [23], the feedback problem is overcome by performing the simulation as the traced program executes. A program driven simulation can be partitioned into a two parts: a memory reference generator, which models the execution of the application and a target system simulator. The target system simulator is the part that models a novel architecture or simply studies the trace generated. An execution driven simulation is performed by instrumenting the application, either at source level or at object level.

We use the program driven approach. The main problem with the trace driven approach, in our case, is that collecting traces make the experiment extremely slow and the trace is in the order of gigabytes for even very short durations of (real time) execution. The execution driven approach often requires source code. Since most applications are commercial in nature this requirement is too restrictive. Execution driven simulation almost always requires that the application be statically linked. Unfortunately, this too is usually not the case because dynamic linking is the preferred approach. In cases where the source is available, rebuilding applications with static linking is not always trivial since it is tough to recreate the build environment.

Our platform for conducting experiments is composed of the following sub-components: the Shade library, Function Information Extractor, Function Store, Recently Used Function Set and Hit Miss History. Fig. 2 shows the interactions between them. At the top level our platform is a loop. Each iteration of the loop simulates one instruction of the program and it can be described by the pseudo code shown in Fig. 3. It checks the program counter(pc) to see whether the pc still points to a location in the function most recently accessed. If that is not the case, the new instruction represents a new reference at the level of functions and the Working Set of functions maintained by the platform is updated, if necessary. The status of the reference (i.e., whether it is a hit or a miss) is stored and the miss rate is computed. All of these actions are initiated by the component labelled *Main* in Fig. 2. It acts like a *driver*, initiating actions which are completed by one of the sub-components. Before the simulated execution of the application under study is begun, its executable is processed to find out information about the functions it is comprised of. The information about each function is stored so that it can be retrieved quickly while the application is being run. In the following subsections the various sub-components are described.

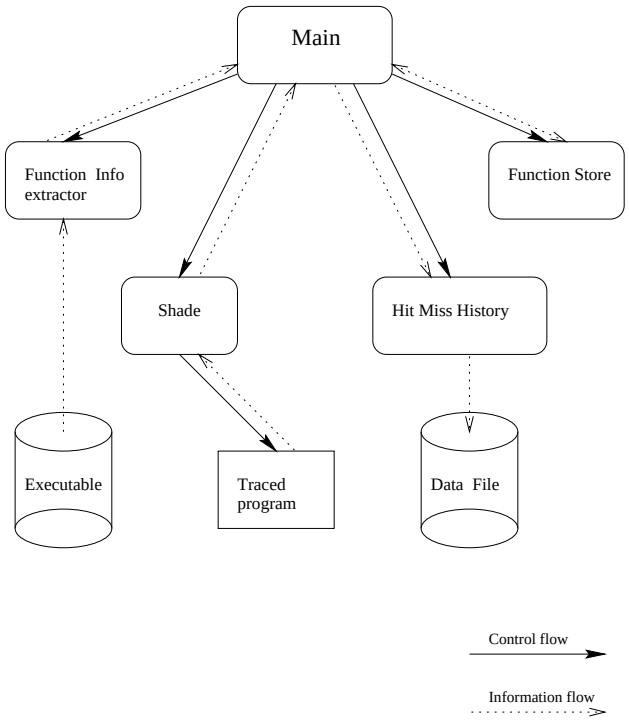

Figure 2: Simulation Methodology

3.1 Shade

Shade [9] is a powerful library of routines, which provide simulation and tracing capabilities. A platform for simulation can be built by implementing an analyzer, which calls various Shade functions to execute an application in a controlled manner and collect the necessary run time information at exactly the points of interest to the user of Shade. The Shade library provides routines to identify the

```
if the address in the program counter is not in the current-function
then
       current-function = function containing the address in program counter
       if (current-function is an element of WS)
       then
              store a hit in the hit-miss-history.
       else
              store a miss in the hit-miss-history.
              if (WS is not full)
              then
                   WS = WS U {current-function}
              else
                   delete the least recently used function from WS.
                   WS = WS U {current-function}
miss rate = number of misses in hit-miss-history / hit-miss-history size
store the miss rate
simulate the next instruction
```

Figure 3: How the main component works

application to be simulated, load the application into the simulated run time environment, run the application, specify the level of detail and the instants at which the tracing information need to be collected during the execution of the program.

The analyzer can request that trace information be collected only for certain machine instructions. It can specify the data, which needs to be collected by specifying the processor registers, memory locations and intermediary results of instruction execution (e.g., the effective address during a control transfer instruction like a branch or a call, whether a conditional branch is taken or not). Moreover, the analyzer can dynamically change the trace information, which needs to be collected, and the instructions for which trace information has to be collected. The analyzer can also specify its own trace functions, which will be called before and/or after simulating an application's machine instruction. These trace functions have access to the state of the simulated machine comprising of the simulated register set and memory locations. The Shade library enables the construction of highly customized trace generators.

The fundamental problem in tracing is that naive methods of building tracing systems result in simulations which slow the program by factors well over 1000 [18]. Running on a sparc and simulating a sparc, shade can execute applications at a slowdown factor well under 10 [9]. Shade achieves this by dynamically building translations for application instructions, caching them for fast access, and chaining them efficiently to amortize simulation and tracing costs

Shade, however, is not a complete machine simulator. The main downside of this is that it limits our study to the user space behavior of a process. When the application makes a system call Shade maps it to a system call on the host operating system. Another drawback is the inability to study multiprogramming. We feel that neither of these factors would significantly alter the conclusions of our study.

3.2 Function Information Extractor

This component provides the functionality for obtaining the byte size and the virtual address of functions. This information is present in an executable's symbol table. Although it is possible to

remove symbolic information from the binaries on Unix platforms, using a utility called *strip*, we found that every executable on our Solaris 5.5.1 platform, which we looked at, did have this information in it.

The binaries on the Solaris platform are in the Executable and Linking Format(ELF). ELF [1] is a framework for object files containing code and data suitable to be linked with other object files, executables files containing programs ready to be loaded, static and dynamic libraries. This format supports multiple data encoding, multiple processors and multiple classes of architectures. An ELF executable contains a header (which identifies the target processor in a machine independent manner), section header table and various kinds of sections. The string table sections contains the different identifiers and the symbol table sections entries for the program symbols. For each function in the executable there is an entry in the symbol table which gives its size and virtual address at which it gets loaded.

The Function Information Extractor uses the *libelf* library to extract this information from the executable. This technique for obtaining information about the functions has the advantage that it does not require the source code, making the study of commercial software feasible. However, the downside to this approach is that we do not obtain information about dynamically linked functions. These functions are part of the shared object files and are not contained in the executable. They get loaded into the virtual address space of the process only at run time. This feature limits our performance evaluation to the portion of code contained in the executable.

3.3 Function Store

The information (size in bytes and its location in the address space) about the functions, which is obtained by the Function Information Extractor from the executable, has to be stored in a way in which it can be retrieved efficiently. During the simulation, for each function, it is also necessary to record whether it has already been accessed in order to distinguish between capacity misses and first reference misses. Hence the need for this component. This is designed as a hash table with a cache. The hash table uses chaining to resolve collisions. To amortize the cost of allocating memory during insertion into a chain, each element of the chain is designed to store information about multiple functions. Thus space is traded off for speed. To increase speed of access, a caching data structure is also supported. The cache is an array with a hashing function. It is "direct-mapped" in the sense that in case of a collision, the older element is discarded.

3.4 Function Working Set

This component keeps track of the working set of functions based on the replacement policy under study. It is used to decide whether the current access to a function results in a hit or a miss. The implementation for the LRU replacement is interesting. The required functionality in this case is the maintenance of a set of objects with an associated access time with the operations:

- Deletemin ( ): Remove the object with the lowest access time from the working set.
- Insert (object, access_time): Insert an object with the specified access time.
- IncreaseAge (object, access_time): If the object specified exists in the working set increase its access time to the specified access time and return success; otherwise return failure.

The implementation of the above Abstract Data Type (ADT) is very similar to that of the Priority Queue ADT in [2, pages 143–145], which provides the first two operations using an array to store all the elements. The third operation, IncreaseAge(), is implemented by doing a linear search followed by an O(log n) reorganization phase.

3.5 Hit/Miss History

This is a circular queue, which remembers the hit/miss outcome of a certain(configurable) number of most recent accesses. It is used in determining

the *capacity miss rate*, i.e., the fraction of capacity misses in the configured window size.

4 Workloads

For this study we chose applications which were not interactive in nature. This was mainly to ensure that our experiments are reproducible. However, many important applications like the editors and browsers fall in this category. Such software is rich in functionality with only a specific portion of their functionality being used at a particular instant of time. For example, editors like emacs have several features like macros, modes for different programming languages, etc., all of which are unlikely to be used even in one specific execution of the program. Our belief is that such software is more likely to exhibit a working set behavior.

The set of applications we studied were gnu awk, unzip, troff, perl and dvips. This was influenced by easy availability. All of them were available in our development environment of Solaris 5.5.1. Ideally we would have liked to study much larger and long running applications like netscape which would have a significant influence on the performance. Besides the fact that many of these are interactive, Shade itself is somewhat restrictive. It is not a perfect simulator and sometimes incomplete Shade functionality has prevented us from choosing a particular application. Another factor, which played an important role in the selection of applications for study, was the number of functions the executable contained, since our hypothesis essentially claims that only a small set of functions gets used over a portion of execution. Therefore we avoided applications containing fewer than 100 functions.

4.1 Awk

Awk – named after its creators Aho, Weinberger, and Kernighan – is a tool for pattern matching and text processing. It processes its input line by line, looking for a pattern match. Every time there is a match it performs an action. An action could involve C like constructs. It acts like a filter; taking its input from standard input, modifying it as specified by an awk script, and printing its output on the standard output. Our workload consists of an awk script which finds C printf statements and prints them. The input is an older version of a C source file used to build the simulator.

4.2 Unzip

Unzip is used to extract files from an archive created by a zip. These tools are widely available across diverse platforms like Unix, Windows NT, Macintosh, OS/2, Atari and MSDOS. The zip archive stores, in general, a collection of files in a compressed form along with all the attributes of the files like modification time, creation time and read/write permissions. It uses the standard format for zip files introduced by Phil Katz. In this workload we extract files from an archive of about 31 kilobytes consisting of 27 text files.

4.3 Troff

Troff is a text formatter for typesetting or laser printing. It allows the users full control over fonts, sizes, and character positions, as well as other features of a formatter like right-margin justification, automatic hyphenation, page titling and numbering. It is normally used through a front end, which translates the higher level formatting commands into troff commands. Our troff workload uses a special command line option to strip off all the troff formatting commands to get an ascii output from a man page on "ct", a communication command on Solaris.

4.4 Perl

Perl is an interpreter for the high level programming language of the same name. It is designed to assist the programmer with common tasks that are probably too heavy or too portability-sensitive for the shell, and yet too weird, short-lived or complicated to code in a language like C. It derives from the C programming language and to a lesser extent from sed, awk and the unix shell. We run a perl

script which finds out the user id, group id, and all the groups to which the user belongs. When the effective user and group ids are different from the real user and group ids, it also prints the effective ids.

4.5 Dvips

Dvips converts files in the standard dvi format (for completely describing a document including the details of the size, style and placement of each character of each page) into the postscript format. We use dvips to convert a draft of an older version of this paper into a postscript file.

5 Observations

In this section we show that there is temporal locality and a working set nature in the references to functions. More specifically, we show that

- caching a small fraction of functions in the program is sufficient to ensure very low miss rates, and
- when misses do occur, they tend to occur together at the same point of time in execution, instead of being distributed uniformly over time.

We also provide complete execution snapshots which show *instantaneous* miss rates as the execution proceeds.

We obtain these results by maintaining a set (of upto a fixed number) of functions in a separate area of memory. This set is initially empty. As the program executes the control flow goes through instructions from different functions. As each function is referenced it is copied into the fixed size working set. On a miss, i.e., a reference to a function which is not already in the set, the LRU policy is used to select the victim function for eviction. Note that successive references to the instructions of the same function do not contribute to different function references. The references to all the instructions in the function, from the point at which the control enters the function to the point at which the control leaves the function, contribute to a single function reference. Thus our results are based on a stricter model of "function reference" – that which eliminates affects of intra-function locality.

Test Program	Num of Funcs	Num of Refs	First Reference Misses
Awk	365	18435	168
Perl	1185	19315	327
Dvips	260	746808	136
Troff	295	56913	110
Unzip	116	27865	75

Table 1: Workload Information

We classify misses as *first reference* and *capacity* misses. First reference misses are those which occur when a function is first accessed. Such misses, at other levels of memory hierarchy, have also been called compulsory misses and cold start misses. Capacity misses are those which occur due to the small capacity of the fixed size working set. They occur when a function which has already been accessed, gets evicted, and has to be retrieved. In general bad replacement policies, a large footprint (compared to the working set size), or bad temporal locality can all contribute to higher capacity misses.

Table 1 gives information about each workload. For each program it specifies the number of functions the executable contains, the number of functions actually referred to during execution (which is equal to the number of compulsory misses) and the reference string length. We observe that the first reference misses are a very small, almost negligible, fraction of the total number of references.

Figure 4 is a plot of the misses against different working set sizes for all the workloads. All graphs decrease as the working set size is increased, since larger working sets should result in fewer misses. In fact, all the graphs have to be monotonically decreasing due to the fact that LRU is a stack algorithm [25, subsection 2.3.2].

All the graphs, except unzip, are concave (or concave up) in nature. This peculiarity, in the case of

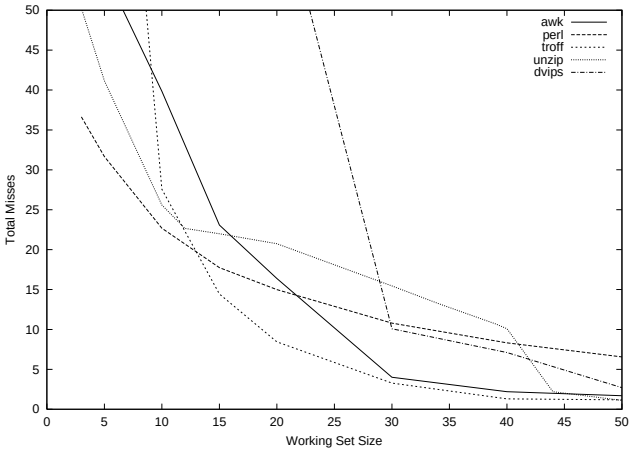

Figure 4: Normalized misses with fixed size working sets

unzip, is explained by the fact that the footprint size of unzip changes from a value less than 12 at one point early in the execution (see Fig 10, where a working set size of 12 is used) to 44 at another point of execution (on further experimentation with unzip we found that it is necessary to increase the working set size beyond 44 to eliminate the thrashing completely). As the working set is increased beyond each of the footprint sizes, many of the capacity misses in the corresponding phase of execution get eliminated and there is a corresponding drop in the total misses.

The misses in fig 4 have been *normalized* by dividing the total number of misses by the number of first reference misses so that the curves for all the workloads can be seen comparatively in a single plot. Thus we see that as the working set size is increased all the graphs settle down to unit value. The plot shows that, in the case of perl, a much larger working set is necessary to make sure that all the misses are first reference misses; This indicates that larger programs are likely to have larger footprints.

In table 2 we have presented the results of studying programs when different fractions of the program size (in terms of number of functions) are used for the working set size. For the purpose of clarity, columns 2 and 3 have been reproduced from table 1. Columns 4 and 5 give the total number of misses and the miss rate, respectively, when a working set of size equal to 10% of the number of functions in the program is used. Miss rate is the percentage of misses in the total number of references. The rest of the columns present the results of using other working set sizes. We observe that, in all cases, a working set size of 40% is sufficient to ensure miss rates of less than 2%. In fact, except in the case of unzip, the miss rate is less than 1% for a 40% working set size. Complete execution snapshots, later in this section, indicate that even unzip requires a large working set only for a certain portion of program execution. If we disregard unzip we observe that a 20% working set size is sufficient for a miss rate of less than 2%. The low miss rates, while caching only a portion of the program, indicate a strong temporal locality in the references to functions.

Figures 5, 6, 7, 8 and 9 show the behavior of all the workloads. On the Y axis we have *capacity miss rate* and the X axis represents a counter whose value gets incremented each time the program counter moves from one function into another. Thus the X axis can be thought of representing virtual time. The clock ticks when the function changes. The capacity miss rate at a particular instant of time can be obtained by looking at a small fixed size interval of time which elapses at the given instant of time, counting the number of capacity misses in that interval, and finding the ratio of the capacity misses to the interval size. The capacity miss rate at a particular instant of time is, thus, an approximation of the derivative of the number of capacity misses with respect to time and measures the number of capacity misses in unit virtual time at the given instant.

We selected the working set sizes of 50, 46, 165, 36 and 44 for dvips, awk, perl, troff and unzip, respectively. These sizes were selected so that a *thrashing* like phenomenon does not set in. A sample plot which depicts thrashing is also provided in figure 10 where a much smaller working set size is used.

Test Program	Num Of Funcs	Num of Refs	WS = 10% of Pgm Size		WS = 20% of Pgm Size		WS = 30% of Pgm Size		WS = 40% of Pgm Size	
			total misses	miss rate(%)	total misses	miss rate(%)	total misses	miss rate(%)	total misses	miss rate(%)
Awk	365	18435	389	2.11	194	1.05	172	0.93	168	0.91
Perl	1185	19315	522	2.70	332	1.72	327	1.69	327	1.69
dvips	260	746808	1999	0.27	334	0.04	154	0.02	136	0.02
Troff	295	56913	477	0.84	122	0.21	113	0.20	110	0.19
Unzip	116	27865	1914	6.90	1474	5.29	995	3.57	139	0.50

Table 2: Shows the total misses and miss rates (in percent) when different fractions of the program size are used as the working set size

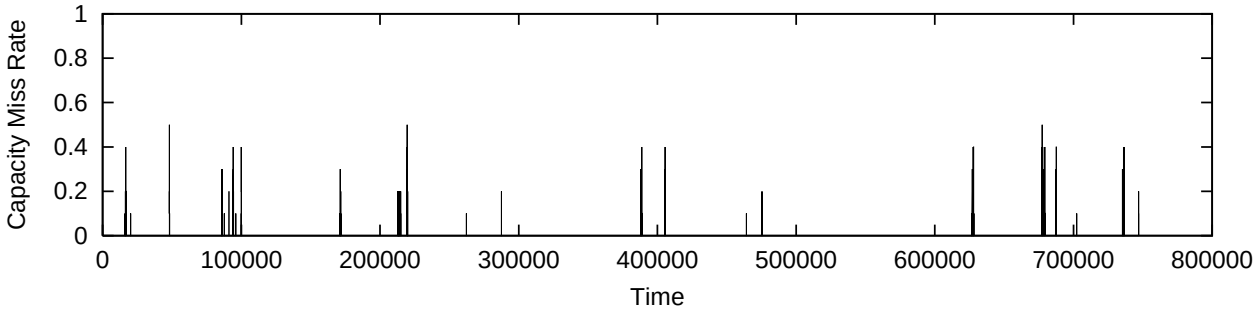

Figure 5: dvips

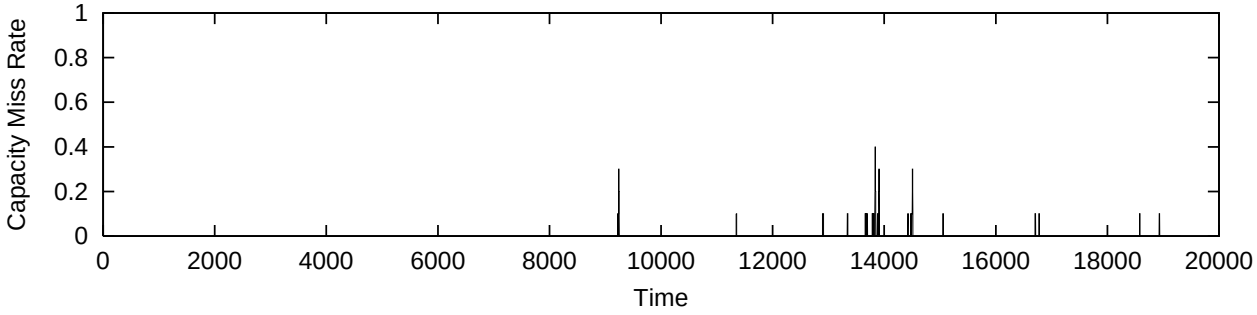

Figure 6: perl

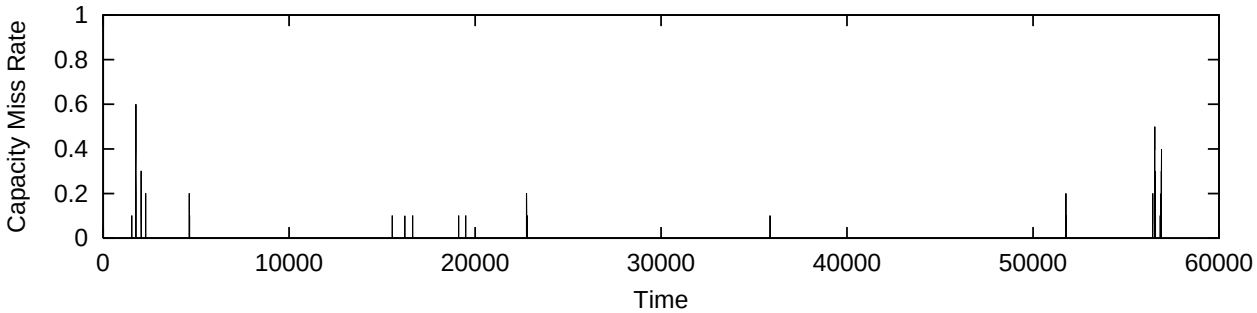

Figure 7: troff

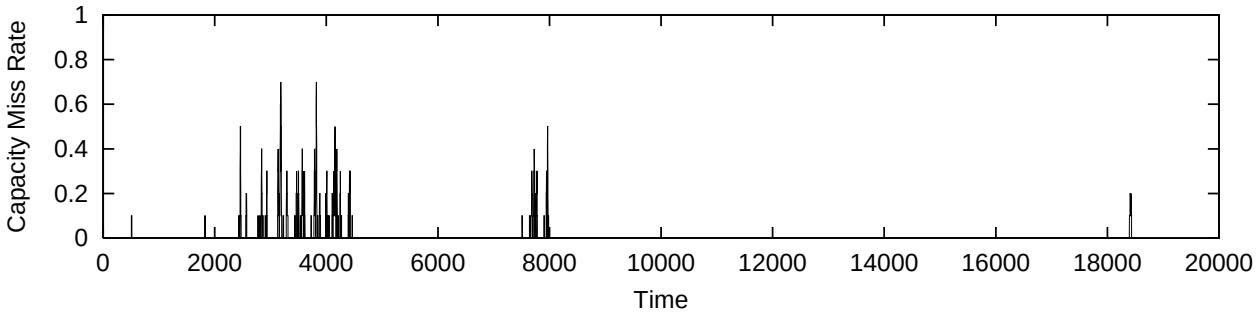

Figure 8: awk

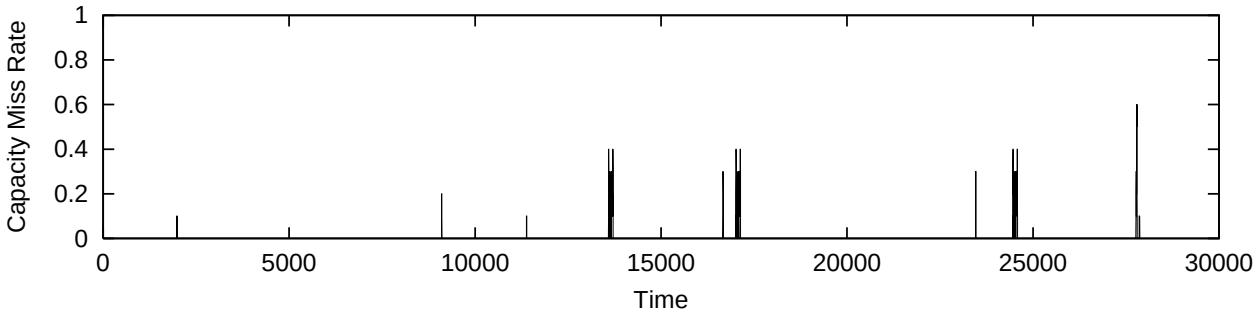

Figure 9: unzip

In each case we notice a working set characteristic. There are very few capacity misses and even those that do occur, occur at around the same point in time. From the fact that there are no capacity misses during most of the execution we can conclude that the first reference misses usually provide a fairly good hint about the functions which would be accessed in the future, implying strong temporal locality in the references to functions. The grouping of capacity misses together indicates that the programs go through different phases of execution, each of which has its own footprint of functions. Figure 10, where a working set size of 12 is used, further illustrates this phase like behavior quite clearly.

In figure 10 there are almost no misses initially, while a certain portion of its execution is characterized by several misses. The initial phase has a small footprint, the later one has a larger footprint and, thus, shows up quite clearly due to thrashing. The working set sizes we have selected for the execution profiles shown in figures 5, 6, 7, 8 and 9 are, therefore, an upper bound of the footprint sizes of different phases of program execution. Otherwise, a thrashing like phenomenon would have shown up in each of those execution profiles. We also note that the working set size selected in each case is bound to be larger than the average footprint size of the corresponding workload. This indicates that it would be beneficial if different programs shared the same caching area, rather than have a separate fixed size cache for each program. The partitioning of this space could be based on some heuristic like the PFF algorithm [7].

In the execution profiles shown in figures 5, 6, 7, 8 and 9 we used a window size of 10 for computing the capacity miss rates. We performed several experiments to see how sensitive our results were to the window size for computing capacity miss rates. In all cases we found that there was no significant impact of the window size on the results obtained. We have provided one sample execution profile in Figure 11, with a different window size of 5 for computing the capacity miss rate, which illustrates this.

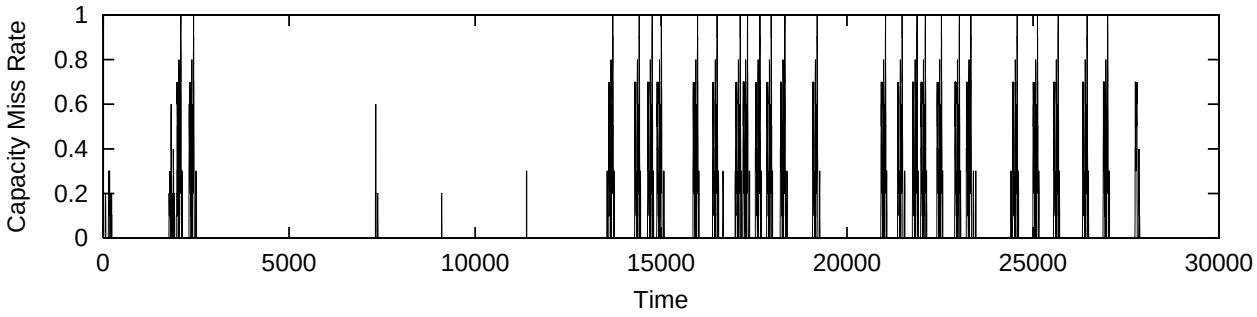

Figure 10: Unzip execution snapshot with a smaller working set size of 12.

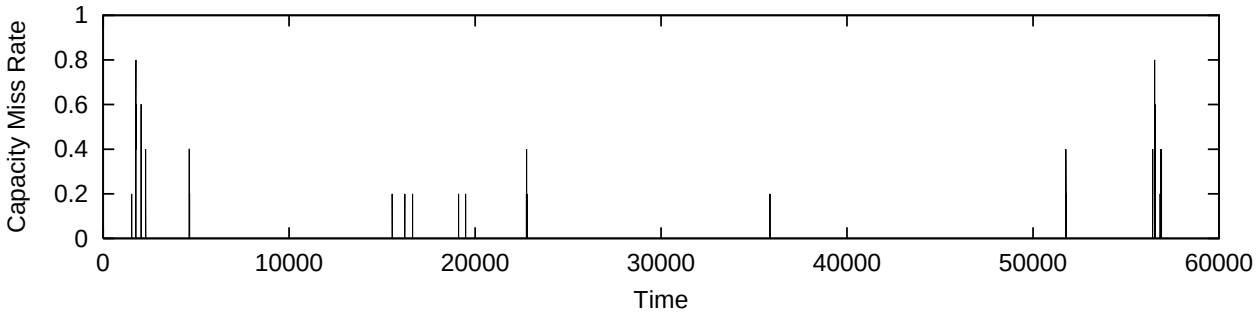

Figure 11: Troff execution snapshot with a different window size to measure the capacity miss rates.

6 Conclusion

In this paper we have shown temporal locality in the references to functions. The complete execution snapshots show capacity miss rates abruptly rising and immediately falling down to zero, indicating a phase like program behavior. The beginning of a new phase is marked with high capacity miss rates. Due to good temporal locality the new functions referred to, get referred in the immediate future, resulting in the miss rate falling down to zero. In most cases, maintaining a working set of 20% of the programs, resulted in complete execution miss rates of less than 2%.

In our study, for simplicity, we fixed the maximum number for functions in the working set. In reality, though, we observed that the number of functions which are used during different phases of a program keep changing. We can take a more adaptive approach in determining how many functions to keep in a working set. One way of doing this is to use the fault rate of functions. In this approach we use a fixed parameter which represents a maximum fault rate. If the actual fault rate during program execution become more than this limit, we do not look for a function to replace. Instead we just increase the working set size to accommodate the new function. Such approaches have been studies earlier at the page level [7].

While we constrained ourselves to use non-interactive and reproducible workloads, we believe that real workloads like editors, browsers and windowing software would have inherently more temporal locality than the workloads we selected. Such software is rich in functionality and typically only a few features are utilized repeatedly, resulting in only a small portion of the code being exercised several times. Software used in real world is, therefore, likely to be more amenable to the exploitation of temporal locality at the function level.

References

[1] Tool interface standards. portable formats specification, version 1.1.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*, chapter 4, pages 143–145. Addison-Wessley, June 1983.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[4] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170. ACM, 1994.

[5] J. Bradley Chen and Bradley D. D. Leupen. Improving instruction locality with Just-In-Time code layout. In *Proceedings of the USENIX Windows NT Workshop*, pages 25–32. USENIX Association, August 1997.

[6] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.

[7] Wesley W. Chu and Holger Opderbeck. The page fault frequency algorithm. In *Proceedings of the 1972 AFIPS Conference*, volume 41, pages 597–609. AFIPS Press, 1972.

[8] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994.

[9] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical Report SMLI 93-12, UWCSE 93-06-06, Sun Microsystems Laboratories, Inc. and University of Washington, 1993.

[10] Edward G. Coffman and Peter J. Denning. *Operating Systems Theory*, chapter 7, pages 286–298. Prentice-Hall, 1973.

[11] Peter J. Denning. The working set model for program behavior. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 323–333. ACM, November 1967.

[12] Peter J. Denning. Virtual memory. *Computing Surveys*, 2(3):153–189, September 1970.

[13] D.J.Hatfield and J.Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[14] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Transaction on Computers*, 50(6):529–548, June 2001.

[15] Nikolas Gloy and Michael D. Smith. Procedure placement using Temporal-Ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, September 1999.

[16] Thomas Kistler and Michael Franz. Continuous program optimization: Design and evaluation. *IEEE Transaction on Computers*, 50(6):549–566, June 2001.

[17] Alexander Klaiber. The technology behind crusoe(tm) processors. Transmeta White Paper, January 2000.

[18] James R. Larus. Efficient Program Tracing. *Computer*, 26(5):52–61, May 1993.

[19] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, 1989.

[20] Sanjay J. Patel and Steven S. Lumetta. rePLay: a hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001.

[21] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[22] Vidyadhar Phalke. *Modeling and Managing Program References in a Memory Hierarchy.* PhD thesis, Rutgers, The State University of New Jersey, October 1995.

[23] R.C.Covington, S.Madala, V.Mehta, J.R.Jump, and J.B.Sinclair. The rice parallel processing testbed. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 4–11, 1988.

[24] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *Proceedings of the IEEE 2001 Workshop on Binary Translation*, September 2001.

[25] Jeffrey R. Spirn. *Program Behavior: Models and Measurements.* Operating and Programming Systems Series. Elsevier North-Holland, Inc., 1977.

[26] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS conference on measurement and modeling of computer systems*, pages 68–79, 1996.